DOCUMENT RESUME

ED 422 938 IR 057 096

AUTHOR	Schrage, John F.
TITLE	Six Thinking Aspects of Programming.
PUB DATE	1997-00-00
NOTE	9p.; In: Proceedings of the International Academy for Information Management Annual Conference (12th, Atlanta, GA, December 12-14, 1997); see IR 057 067.
PUB TYPE	Information Analyses (070) -- Speeches/Meeting Papers (150)
EDRS PRICE	MF01/PC01 Plus Postage.
DESCRIPTORS	Academic Achievement; Computer Oriented Programs; *Computer Science Education; *Computer Software Development; Critical Thinking; Design Requirements; Higher Education; Programmers; *Programming; Teaching Methods; *Thinking Skills

ABSTRACT

Based on literature and student input, six major concerns have been noted for student programming progress for the academic class and work environment. The areas of concern are module driver programming, program documentation, output design, data design, data validation; and reusable code. Each area has been analyzed and examined in the teaching of computer programming over a period of about 20 years. The key element continues to be thinking. Getting programming students to think about each of the concerns and applying those principles to their environment leads to a better programmer. (Contains 15 references.) (Author/AEF)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

SIX THINKING ASPECTS OF PROGRAMMING

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☐ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

John F. Schrage
Southern Illinois University at Edwardsville

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
T. Case
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

ED 422 938

Based of literature and student input, six major concerns have been noted for student programming progress for the academic class and work environment. The areas of concern are module driver programming, program documentation, output and data design, data validation, and reusable code. Each area has been analyzed and examined in the teaching of computer programming over a period of about twenty years. The key element continues to be thinking. Getting programming students to think about each of the concerns and applying those principles to their environment leads to a better programmer.

INTRODUCTION

After reviewing student's comments from course evaluations on the programming class for over 20 years, a continuing concern regarding thinking prevails. In addition to the normal course evaluation, the author continues to periodically evaluate class topics in the programming course to determine clarity of presentation and to follow-up student questions or lack thereof in the class. Students believe that the major effort for the programming course is getting the correct answer to the programming problem. But life normally does not have just one answer and that mirroring is not realistic. While the concerns expressed from students relate to the COBOL programming course, much of the material is tangential to almost any language. Even in home page design and its coding, the relationship of the concerns has been noted by and to students. Programming-related concerns have been noted by students and thus should be addressed more by faculty in the process of teaching programming courses – no matter what language is used. Over the last twenty years, six major concerns continue to be expressed by students over the last twenty years have been the following:

a. module driver programming
b. program documentation
c. output design
e. data design
e. data validation
f. reusable code

In the first years of teaching, the faculty member normally leans heavily on the textbook and its ancillaries. As teaching time passes, the faculty member should be aware of the fact that books change but many of the programming constructs continue. The textbook provides one view in the logical progression on programming plus examples for the student to follow in that progression and finally situations to program which illustrate the progression of topics. The programming assignment should not just be a coding assignment but a means for the student to think about results for the user and what really is involved in the process. This is leading to a more realism, which seems to be a missing element for the student. No matter which textbook is used, the faculty member should be providing an instruction mode to student learning not just lecturing from a book.

MODULE DRIVER PROGRAMMING

One of the first concerns given by students centered on the aspect of structured programming, which leads to the module mode of programming and thus the driver programming method. GO-TO programming was the way programming was initially taught. Then structured programming made the GO-TO verb fade – in BASIC, COBOL, dBASE and _____ (insert your choice of language). Programming in the middle of the seventies was starting to emphasize the structured approach and changed the process of programming. An experienced

IR057096

programmer taking an academic course generally fought with the faculty member who used the structured approach based on the process of "getting the program completed". The author vividly remembers arguments with working students who would argue about being able to code the program faster by ignoring structured concepts. Coding the program was easy but the thinking part really caused students some concerns. Over time structured methods prevailed and thus the module driver approach appeared. In the sixties, the author remembers the COBOL programming course in which four programs were the normal number written for the whole course. When the structured methods were applied much more coding could be accomplished. The applied theory of the New York Times programming project directly effected the academic expectation of programming. With the driver method in COBOL, the academic level of programming approached at least 30 statements per day per credit hour production.

In pushing students, two studies were attempted to note programming prowness. For about two years, a now retired faculty member and the author controlled the programming level of students using the academic quarter (10 weeks with 4 fifty-minute meetings weekly). The initial failure rate in classes was high based on students not allocating the expected time for class as requested by academic theory – at least one hour of outside class time devoted to the class for every hour of class. After students accepted the time expectation in programming, the passing rate climbed. In fact, the "A" grade rate was so high that questions arose on giving grades in the class. Students were most inclined to complete the work at a high level based on the expected first job being in COBOL programming. In the last three years, the grades have come down, based mainly on students finding out that programming required much thinking and also the fact that companies were hiring less programmers. In the last year, the intensive COBOL seminar approach has increased based on the lack of COBOL programmers and the year 2000 project.

In teaching the module and driver approaches, the student had to think in a manner that the whole is made up of several parts, and thus the student tried to solve programs in pieces. The building module approach has again increased the level of student coding with the number being between 30-50 statements per day. This was not the only reason for the programming increase but seemed to be the root of the solution. The microcomputer with its availability of compilers has also played a part in productivity.

With the driver approach, the student better organized his or her thoughts to come to a conclusion. The author provides a series of generic programs for the whole process which follow a building approach for listing, calculations, totals, page breaks, conditional logic, control breaks, sorting, tables, and screens. The course problems also lean heavily on the building approach but requires design thinking for the student. [A sample set of programs noted in this section is available in the public domain.]

PROGRAM DOCUMENTATION

The author brings into class what is left of a 50-page program of the early seventies with no documentation and asks several questions of the students in the class about the program after the initial programming lectures and first student program. The general student comments deal with the junkiness of the production program, which came from a Fortune 500 company. Program documentation should be emphasized in any class in which some level of programming is attempted. Program documentation includes not only internal comments but data name construction rules for field and calculation clarity.

Even when HTML is discussed in the creation of home pages, the program comment aspect is explained and required to aid the student in further refinement of the code – even when most of the class are not computing majors. As a part of any programming class, program documentation should be emphasized. In reviewing over 167 course outlines from an Internet search, only twenty-seven (27) outlines had the topic of documentation as part of lecture materials and six (6) outlines had an internal document on standards that should be applied to programming assignments. From work experiences with the US Army, their Computer Science School previously provided a fifteen-page document on standards and program documentation. Dr. Janet Hartman of Illinois State University had standards on her Web page for her C programming class. From other

research and faculty interaction on the topic, Dr. R. Wayne Headrick of New Mexico State University, has a programming standards handout, which details program documentation and COBOL coding standards, which is also, noted on his course web pages.

For COBOL, minimum standards include general programming conventions, internal program comments, spacing, data name conventions, and module name conventions. Some of the general programming conventions include:

IDENTIFICATION and ENVIRONMENT DIVISION considerations:

a. IDENTIFICATION and ENVIRONMENT DIVISIONS can be placed on separate pages by placing a / in column 7.

b. The sequence in which files are selected in the SELECT statements is not critical, but the more logical approach is to select and define the input file(s) first and then the output file(s).

c. Code each SELECT statement on two lines and avoid the use of device-specific file-names such as SALES-DISK.

PROCEDURE DIVISION considerations:

a. A Mainline or Driver module should control execution of the program modules in the PROCEDURE DIVISION. That module should be the first module in the PROCEDURE DIVISION.

b. Each program module should include only the statements required to accomplish a process.

c. Each READ and WRITE statement required by the program should be coded into a separate program module.

d. STOP RUN should be the last executed statement in the Mainline or Driver module.

e. Only code a single statement per line for the sake of clarity and to make debugging easier.

Internal Program Comment Requirements:

a. Immediately following the IDENTIFICATION DIVISION, there should be comments that indicate the overall function of the program. So that the comments can be easily identifiable as documentation text, proceed and follow the comments with a line of *s.

b. With the exception of the Mainline and Initialization modules, all program modules in the PROCEDURE DIVISION must be proceeded by comments that indicate the purpose of the module.

Spacing Requirements:

a. All indentation should be made in increments of four spaces.

b. All PIC and VALUE clauses must be aligned.

c. If a COBOL sentence is too long to fit on a single line, the second line must be indented four spaces.

d. All "blank" lines should have an * in the 7th column with the rest of the line being blank.

e. All division headers, with the exception of the IDENTIFICATION DIVISION should be proceeded by two or three "blank" lines.

f. All section headers should be proceeded by one or two "blank" lines and followed by one "blank" line.

g. All file descriptions (FDs) should be proceeded by one "blank" line.

h. All 01-level record descriptions should be proceeded by one "blank" line.

i. All PROCEDURE DIVISION program module names should be proceeded by one "blank" line and followed by one "blank" line.

Data Record Naming Conventions:

a. All program accumulation, computation and control fields should have a prefix or suffix of WS. Temporary fields of this nature should have similar fields grouped together at the 01 level, then each individual field listed at the 05 level.

b. All input data record and fields should have a prefix or suffix to indicate input [I or IN] followed by all output print records and fields should have a prefix or suffix to indicate output [O or OUT]. In the same manner, all heading records and fields should have a prefix or suffix that begins with the letter H followed by a number indicating which heading line is specified.

Module Naming Conventions and General Considerations:

a. The module name should indicate the specific function that the module is performing.

b. Code module names on a line by themselves.

c. Module names should be composed of a one-word verb followed by a two-word object.

d. A naming convention should be used to assign an appropriate 3 digit prefix to the module name, such as:

e.

Module #	Module Function
000	Mainline
025-090	Sort & Other General Items before Initialization
100-199	Initialization Processing
200-299	General Processing
300-399	End-of-Run Statistics
500-599	Input Processing
600-649	Output Processing
650-699	Headings
700-749	Page Processing
750-799	Called Programs
890-899	Detail Line Processing
999	Last Line paragraph

The above noted COBOL programming standards and documentation items should be applied to every programming course. Many students do not see the need for the rules in the first set of elementary program but then a light seems to grow brighter on why the rules are applied to the computer programs.

OUTPUT DESIGN

The output is what the user wants to view to be able to make accurate decisions for company actions. Based on the output, the programmer must determine how the data must be manipulated to produce that expected set of output. Even in the beginning classes, the author has the student design output. Whenever an output design is given to the students, a mirroring of that output has been accomplished without much problem. When the output becomes variable or designed by the student, then problems seem to happen.

The author, in trying to get students to think, has the student design the actual output for all programs in the beginning class, based on specifications provided. Initially in just telling student which fields should be on the output report, the student has to space the headings and data in a useable manner. Rules are provided on the general paper design and then students layout the headings and fields. The basic output format is presented to include company and report titles, field headings, data detail lines, total lines, and end of report indicators plus form specification notation. Based on questioning students on the first report format assignment, over seventy percent worried more on the design issue than the coding of the application. In the advanced programming course, the first problem deals with three small programs – formatted report, data output dump, and audit report. While the logic and code is as simple as the first program written in the beginning class, about seventy-five percent of the students struggle with the output design issue with special concerns about the audit report – which is noted as first record, last record, nth record(s), and some numeric totals or limitations.

At the beginning, the one-page reports provide a start to the whole design process for user expectations. The use then of multiple headings and detail lines, then selected detail lines, and multiple output design are just interations of the primary output principle but various thinking problems arise. Not all students have problems but a majority has concerns until the thinking light starts to lead the student. Integrated with the module material are concepts of deviation of the basic output to add such items as multiple output lines, page routines, and total lines.

The use of screen design is presented for another output design mode. Just as with paper output, a screen design template is discussed with a basic structure. That structure is enhanced in various programming courses depending on the various options that can be used in a normal manner. Statistically, the trend for screen design concerns has been lessening over the past five years. When students are asked why screens seem easier, the major answer is related to microcomputer use and particularly the use of word processing principles plus the relationship to use of presentation graphics in classes. A new aspect of screen design is being examined in respect to the graphical interface presented with windows and the multiple entities such as drop

down list boxes which are available in Visual BASIC and also for other languages.

DATA DESIGN

For the person that thinks that output design causes some concerns on student thinking, try data design. With a systems class and programming class required for the advanced programming course, the author initially thought that students would remember basic concepts on the easy assignment of creating student-based data for the problem. Data design blows away more students that the other aspects previously noted. For the faculty member that does not agree with this, you should design your own programs for students to solve each time the programming class is presented. As noted previously, the author does not teach from a particular textbook but emphasizes content and references the textbook. Programs from previous terms seem to circulate in the academic community and thus there does not seem to be enough of the textbook problems.

The actual design of data was a function of the advanced programming course for over ten years. When the author went back to teaching the beginning class, the data design issue was tried in the beginning class with huge failure. The recovery on data design hampered the normal coverage of programming expectation in the first programming course. The particular time in which the task was attempted did not matter. Students without the first systems class really struggled with data design. Thus other avenues for data acquisition for the first programming class are now being used.

Data design for the author has come from two directions–students in the advanced programming course and a very good graduate assistant with work experiences. A third mode was used for several years but the application textbooks disappeared with the consolidation of publishers. Alan Eliason at Oregon State University had several editions of his business applications book on the market over the past twenty years and the material was excellent for showing data flow and application aspects.

The advanced programming course had students assigned applications in which file processing and updating were the main focus. All of the beginning class concepts were integrated such that the application could, with minor modification, be used in the first programming class. With the application book and various articles, students designed the file structure and then created actual data in the file-based course. Again a thinking process in the programming course. Thirty-five application areas were provided to the student with additional areas negotiated between the student and instructor. The application areas were noted in the reference index titled, Computer Literature Index, which was published quarterly.

The author also had an information systems graduate student design a new data set in 1995 after reviewing the assignments given to the classes for about seven years. The new case design include: hotel management, television programming with marketing, hardware store inventory, customer tracking for a mail order catalogue, and beer distributor. The data sets are now being revised again and some new sets designed for use in other programming courses taught by the author. On analyzing student achievement in the course, the best results from students have been with the beer distributor application–something that interests some students in spare/social moments.

If after the above design concerns, the instructor still want students to design the data, what approach can be used? The author had the least problems when the file structure was provided to the student. Given the fields and their characteristics, data can be extracted from various sources. The telephone book and various catalogues are the best sources of data content. Even when the author has used this approach, students seem to get stuck in providing realistic data after about ten to twelve records. The main success to data design seems to be related to the maturity level of the student and his/her previous work experiences. Thus, if you are teaching a traditional student, data design can be a concern. If you have older students with work experiences, the data design aspect is even fun for the student, based on student feedback.

DATA VALIDATION

Controls are obviously needed for any application no matter how processed. Errors occur in the processing of the data with the severity

dependent on the dollar relationship between the error and its results. The nature of each error, from the bit to the file usage, can be traced to specific sources. The cause frequently implies the method of choice for correcting that situation. Man, in a generic sense, has always used his thinking ability coupled with business knowledge to control and edit the manual process. These same aspects must also be used in machine processing. The computer program can methodically edit and control data for applications, even to including examining character bit structures, but only in so far as human instructions are provided to the machine.

The organization should be examining all data input/entry within every application such that a minimization of errors affects the monetary structure of the organization. Programmers should be implementing general edit techniques in all application areas. Surveying the normal instructional procedures by which a beginning programmer is taught data validation programming and processing has given some concern. Changes to data validation appears to be happening in those organizations, which emphasize interactive use of data.

Some languages have added validation keywords to the language structure to aid programming. The Paradox package allows validation of fields with limits (high and low) and the BASIC language has built-in validation of numeric data for numeric fields. Even the graphical client/server application development tool of PowerBuilder has data validation rules that can be coded in the Database painter or the DataWindow painter. The COBOL 97 standard should have a new verb called VALIDATE, which will allow the programmer to check the field without writing several lines of code. Many languages, such as COBOL, FORTRAN, C++, and C, provide for data validation by the "brute force" mode of checking fields or character by character into a buffer mode. While the data validation techniques seem to be taught in many academic programs and business training settings, what is taught and what is implemented seem to differ.

Thirty businesses in one geographic area were analyzed using a validation technique matrix (which is shown in the appendix). Only twenty-two of the businesses were using some technique. Twelve businesses were totally validating data as entered into the system before processing. One major financial institution did not validate any data but expected the data entry people not make mistakes. The absence of validation was treated very lightly. The comment from the controller was that, "no errors have occurred which have been financially adverse to the company and the insurance firm for the organization would handle any losses". Based on the survey, twenty-seven firms decided to re-evaluate their data validation process. The author spent two working days with three companies to examine validation. In the stay, data errors, which were accepted by the system ranged from one error for 500 fields, entered to three errors in 120 fields in another firm. Thus firms which validate data with the software still can allow an error to be entered into the system. No system is totally fool proof in data validation. Programming code was examined in all cases to determine how errors were caught and how errors were able to get into the system. Only with extensive testing on the part of both the programming staff and the user community can errors be kept to an absolute minimum. Those errors, which still were able to get into the system, were not normal entities entered into the application but some data items, which we were trying to input to get the system to fail.

Many programming-related materials have tended to minimize the aspects of data validation to concentrate on other programming specific concepts. The current business-oriented computer curricula of DPMA and ACM [and even the new proposed joint curriculum between ACM, DPMA, and AIS] implies the need for data validations but seems to treat the concept as learned in the academic environment and reinforced in training programs. Most programming-oriented training seems to imply the need for data validation but expects the programmer trainee to know what should be done. Data validation and other controls are essential for the preventative medicine aspects of information processing. When educators are teaching the programming sequence, bad data must be used to alert the student to the real world rather than just concentrating on mirroring the output.

REUSABLE CODE

Reusable code is the newest student issue. The original program coding and subsequent re-coding of revisions is being done less but a code library must be present for the reuse issue to be used properly. The use of the COBOL COPY verb is discussed during the lecture material on formatting output for the second programming assignment. The notation that file structures are provided by the data base administrator and then copied into the program rather than being entered for each program by the programmer is emphasized. While not required to be used in that second assignment, over eighty percent of the students take the hint to use the COPY verb to bring in the file description. The same copying concept is noted in the third assignment with constants, page controls, report title, and end-of-report expectations. Again, the use of the copy verb is used by about fifty percent of the students. The author relates his experience in designing label producing programs with one to five up labels and the coding issue. Students then design a label program with these principles in mind. The program is then exchanged with other classmates to produce a class-based label program, which can be used in other assignments.

As noted earlier in the article, a set of generic programs is used in the lectures on concepts. A generic COBOL program is initially presented to show the parts to the program and the relationship of reserved word and programmer-supplied data names. A series of programs then use that generic code to present the course concepts of data dumping, format report, calculations, page insertions, conditional testing, control breaks, sorting, multiple page breaks, tables (internal and external), and screens. Those programs thus become the COBOL library for students to use and modify. While that set is available, many students still code each program from scratch. Modification are being made to the course such that a student will have to use selected programs from this generic library to complete assignments in a timely manner.

In teaching home page design, students are given a generic structure for a resume and then adapt that code to their setting. Many students search the Internet for selected backgrounds, graphics, and page formats to design pages for themselves. That correlation to reusable code is understood by students more than as presented in the normal programming course, but the point of using and re-using code for whatever language is being done more. A fine line is presented in the re-using concept and copying code between students for assignments.

With the reduction of programmers and the use of design tools, the need for original code will continue to decrease but only to the extent that the software library grows with examples of the coding procedures. The code libraries for Pascal, C, C++, and Visual BASIC continue to grow in the shareware area but little has been provided for the COBOL language.

CONCLUSION

While emphasizing the above six topics will not guarantee a good programmer, the author has received much positive feedback from local area companies and former students who have been through the process. The initial "survival" parties have disappeared but the concepts are continually asked in the interview process. While the programming topics have lessened, based on the changing nature of the information system student product, the carryover of these principles are emphasized in the data base, data communication, and systems courses. Using the above items does put more pressure on both the student and the instructor.

Thinking aspects are not just one direction. In programming, the thinking process is two way and should be made as interesting as possible for both the faculty member and the students enduring the process of coding in the variety of languages taught in the academic setting. The first experience of the author using computers was almost thirty years ago as a college senior in a traditional business computer concept course, which included FORTRAN programming. That spark led to further courses in graduate school and eventually teaching the material many times during the last quarter century. In many ways, the computer programming material reminds the author of the first college accounting course taught. The senior faculty members helped me get organized as I was taking over for another faculty member who became ill. The senior

accountant told me to remember that the concepts stay the same over time plus will lessen in importance but thinking will always provide the challenge.

Note: For a copy of the diskette with sample programs, documentation standards, and PowerPoint slides, e-mail the author for reply with a compressed file attachment.

REFERENCES

Charlton, John. "The Great Divide", *Computer Weekly*, May 9, 1996, pages 38-40.

Dahlstrand, Ingemar. Software *Portability and Standards*. West Sussex, England: Ellis Horwood Limited, 1984, 150 pages.

Dunn, Robert H. *Software Quality: Concepts and Plans*. Englewood Cliffs, NJ: Prentice-Hall, 1990, 296 pages.

Grauer, Robert T. *A COBOL Book of Practice and Reference*. Englewood Cliffs, NJ: Prentice-Hall, 1981, 382 pages.

Humphrey, Watts S. *Introduction to the Personal Software Process*. Boston: Addison-Wesley Longman, Inc., 1997, 278 pages.

Miller, Philip L and Lee W. Miller. *Programming by Design*. Belmont, CA: Wadsworth Publishing Company, 1986, 567 pages.

Noll, Paul. *Structured COBOL Methods*. Fresno, CA: Mike Murach and Associates, 1997, 208 pages.

Philippakis, A.S. and Leonard J. Kazmier. *Program Design Concepts with Application in COBOL*. New York: McGraw-Hill Book Company, 1983, 217 pages.

Robertson, Lesley A. *Simple Program Design, second edition*. Danvers, MA: Boyd and Fraser Publishing Company, 1993, 203 pages.

Satzinger, John W. and Tore U. Orvik. *The Object-Oriented Approach: Concepts, Modeling, and Systems Development*. Danvers, MA: Boyd and Fraser Publishing Company, 1996, 163 pages.

Seybold, Patricia. "Replace Waterfall Methods with Workflow Methods", *Computerworld*, May 23, 1994, 28:21, page 37.

Shultz, Steven S., Joel A. Farrell, and Vladimir R. Yakhoris. "Deriving Programs using Generic Algorithms", *IBM Systems Journal*, March 1994, 33:1, pages 158-82.

Topper, Andrew. *Object-Oriented Development in COBOL*. McGraw-Hill, Inc., 1995, 487 pages.

Vesely, Eric G. *COBOL: A Guide to Structured, Portable, Maintainable, and Efficient Program Design*. Englewood Cliffs, NJ: Prentice-Hall, 1989, 410 pages.

Young, Peter. "Australian Software Methods may Become International Standards", *Computerworld*, June 27, 1994, 28:26, page 116.